Flask-SocketIO Documentation

Release

Miguel Grinberg

May 09, 2016

Contents

Flask-SocketIO gives Flask applications access to low latency bi-directional communications between the clients and the server. The client-side application can use any of the SocketIO official clients libraries in Javascript, C++, Java and Swift, or any compatible client to establish a permanent connection to the server.

CHAPTER 1

Installation

You can install this package in the usual way using `pip`:

```
pip install flask-socketio
```

CHAPTER 2

Requirements

Since version 1.0, this extension is compatible with both Python 2.7 and Python 3.3+. The asynchronous services that this package relies on can be selected among three choices:

- eventlet is the best performant option, with support for long-polling and WebSocket transports.
- gevent is the framework used in previous releases of this extension. The long-polling transport is fully supported. To add support for WebSocket, the gevent-websocket package must be installed as well. The use of gevent and gevent-websocket is also a performant option, but slightly lower than eventlet.
- The Flask development server based on Werkzeug can be used as well, with the caveat that it lacks the performance of the other two options, so it should only be used to simplify the development workflow. This option only supports the long-polling transport.

The extension automatically detects which asynchronous framework to use based on what is installed. Preference is given to eventlet, followed by gevent. If neither one is installed, then the Flask development server is used.

If using multiple processes, a message queue service is used by the processes to coordinate operations such as broadcasting. The supported queues are Redis, RabbitMQ, and any other message queues supported by the Kombu package.

On the client-side, the official Socket.IO Javascript client library can be used to establish a connection to the server. There are also official clients written in Swift, Java and C++. Unofficial clients may also work, as long as they implement the Socket.IO protocol.

2.1 Differences With Flask-SocketIO Versions 0.x

Older versions of Flask-SocketIO had a completely different set of requirements. Those old versions had a dependency on gevent-socketio and gevent-websocket, which are not required in release 1.0.

In spite of the change in dependencies, there aren't many significant changes introduced in version 1.0. Below is a detailed list of the actual differences:

- Release 1.0 drops support for Python 2.6, and adds support for Python 3.3, Python 3.4, and pypy.
- Releases 0.x required an old version of the Socket.IO Javascript client. Starting with release 1.0, the current releases of Socket.IO and Engine.IO are supported. Releases of the Socket.IO client prior to 1.0 are no supported. The Swift and C++ official Socket.IO clients are now supported as well.
- The 0.x releases depended on gevent, gevent-socketio and gevent-websocket. In release 1.0 gevent-socketio is not used anymore, and gevent is one of three options for backend web server, with eventlet and any regular multi-threaded WSGI server, including Flask's development web server.
- The Socket.IO server options have changed in release 1.0. They can be provided in the SocketIO constructor, or in the `run()` call. The options provided in these two are merged before they are used.

- The 0.x releases exposed the gevent-socketio connection as `request.namespace`. In release 1.0 this is not available anymore. The request object defines `request.namespace` as the name of the namespace being handled, and adds `request.sid`, defined as the unique session ID for the client connection, and `request.event`, which contains the event name and arguments.
- To get the list of rooms a client was in the 0.x release required the application to use a private structure of gevent-socketio, with the expression `request.namespace.rooms`. This is not available in release 1.0, which includes a proper `rooms()` function.
- The recommended "trick" to send a message to an individual client was to put each client in a separate room, then address messages to the desired room. This was formalized in release 1.0, where clients are assigned a room automatically when they connect.
- The `'connect'` event for the global namespace did not fire on releases prior to 1.0. This has been fixed and now this event fires as expected.
- Support for client-side callbacks was introduced in release 1.0.

2.2 Upgrading to Flask-SocketIO 1.x and 2.x from older releases

On the client side, you need to upgrade your Socket.IO Javascript client from the 0.9.x releases to the 1.3.x or newer releases.

On the server side, there are a few points to consider:

- If you wish to continue using gevent, then uninstall gevent-socketio from your virtual environment, as this package is not used anymore and may collide with its replacement, python-socketio.
- If you want to have slightly better performance and stability, then it is recommended that you switch to eventlet. To do this, uninstall gevent, gevent-socketio and gevent-websocket, and install eventlet.
- If your application uses monkey patching and you switched to eventlet, call *eventlet.monkey_patch()* instead of gevent's *monkey.patch_all()*. Also, any calls to gevent must be replaced with equivalent calls to eventlet.
- Any uses of *request.namespace* must be replaced with direct calls into the Flask-SocketIO functions. For example, *request.namespace.rooms* must be replaced with the *rooms()* function.
- Any uses of internal gevent-socketio objects must be removed, as this package is not a dependency anymore.

CHAPTER 3

Initialization

The following code example shows how to add Flask-SocketIO to a Flask application:

```
from flask import Flask, render_template
from flask_socketio import SocketIO

app = Flask(__name__)
app.config['SECRET_KEY'] = 'secret!'
socketio = SocketIO(app)

if __name__ == '__main__':
    socketio.run(app)
```

The `init_app()` style of initialization is also supported. Note the way the web server is started. The `socketio.run()` function encapsulates the start up of the web server and replaces the standard `app.run()` standard Flask development server start up. When the application is in debug mode the Werkzeug development server is still used and configured properly inside `socketio.run()`. In production mode the eventlet web server is used if available, else the gevent web server is used. If eventlet and gevent are not installed, the Werkzeug development web server is used.

The application must serve a page to the client that loads the Socket.IO library and establishes a connection:

```
<script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/socket.io/1.3.6/socket.io.min.js
<script type="text/javascript" charset="utf-8">
    var socket = io.connect('http://' + document.domain + ':' + location.port);
    socket.on('connect', function() {
        socket.emit('my event', {data: 'I\'m connected!'});
    });
</script>
```

CHAPTER 4

Receiving Messages

When using SocketIO, messages are received by both parties as events. On the client side Javascript callbacks are used. With Flask-SocketIO the server needs to register handlers for these events, similarly to how routes are handled by view functions.

The following example creates a server-side event handler for an unnamed event:

```
@socketio.on('message')
def handle_message(message):
    print('received message: ' + message)
```

The above example uses string messages. Another type of unnamed events use JSON data:

```
@socketio.on('json')
def handle_json(json):
    print('received json: ' + str(json))
```

The most flexible type of event uses custom event names. The message data for these events can be string, bytes, int, or JSON:

```
@socketio.on('my event')
def handle_my_custom_event(json):
    print('received json: ' + str(json))
```

Custom named events can also support multiple arguments:

```
@socketio.on('my event')
def handle_my_custom_event(arg1, arg2, arg3):
    print('received args: ' + arg1 + arg2 + arg3)
```

Named events are the most flexible, as they eliminate the need to include additional metadata to describe the message type.

Flask-SocketIO also supports SocketIO namespaces, which allow the client to multiplex several independent connections on the same physical socket:

```
@socketio.on('my event', namespace='/test')
def handle_my_custom_namespace_event(json):
    print('received json: ' + str(json))
```

When a namespace is not specified a default global namespace with the name `'/'` is used.

Clients may request an acknowledgement callback that confirms receipt of a message. Any values returned from the handler function will be passed to the client as arguments in the callback function:

```
@socketio.on('my event')
def handle_my_custom_event(json):
    print('received json: ' + str(json))
    return 'one', 2
```

In the above example, the client callback function will be invoked with two arguments, `'one'` and `2`. If a handler function does not return any vallues, the client callback function will be invoked without arguments.

CHAPTER 5

Sending Messages

SocketIO event handlers defined as shown in the previous section can send reply messages to the connected client using the `send()` and `emit()` functions.

The following examples bounce received events back to the client that sent them:

```
from flask_socketio import send, emit

@socketio.on('message')
def handle_message(message):
    send(message)

@socketio.on('json')
def handle_json(json):
    send(json, json=True)

@socketio.on('my event')
def handle_my_custom_event(json):
    emit('my response', json)
```

Note how `send()` and `emit()` are used for unnamed and named events respectively.

When working with namespaces, `send()` and `emit()` use the namespace of the incoming message by default. A different namespace can be specified with the optional `namespace` argument:

```
@socketio.on('message')
def handle_message(message):
    send(message, namespace='/chat')

@socketio.on('my event')
def handle_my_custom_event(json):
    emit('my response', json, namespace='/chat')
```

To send an event with multiple arguments, send a tuple:

```
@socketio.on('my event')
def handle_my_custom_event(json):
    emit('my response', ('foo', 'bar', json), namespace='/chat')
```

SocketIO supports acknowledgement callbacks that confirm that a message was received by the client:

```
def ack():
    print 'message was received!'

@socketio.on('my event')
```

```
def handle_my_custom_event(json):
    emit('my response', json, callback=ack)
```

When using callbacks the Javascript client receives a callback function to invoke upon receipt of the message. After the client application invokes the callback function the server invokes the corresponding server-side callback. If the client-side callback returns any values, these are provided as arguments to the server-side callback.

The client application can also request an acknoledgement callback for an event sent to the server. If the server wants to provide arguments for this callback, it must return them from the event handler function:

```
@socketio.on('my event')
def handle_my_custom_event(json):
    # ... handle the event

    return 'foo', 'bar', 123  # client callback will receive these 3 arguments
```

CHAPTER 6

Broadcasting

Another very useful feature of SocketIO is the broadcasting of messages. Flask-SocketIO supports this feature with the `broadcast=True` optional argument to `send()` and `emit()`:

```
@socketio.on('my event')
def handle_my_custom_event(data):
    emit('my response', data, broadcast=True)
```

When a message is sent with the broadcast option enabled, all clients connected to the namespace receive it, including the sender. When namespaces are not used, the clients connected to the global namespace receive the message. Note that callbacks are not invoked for broadcast messages.

In all the examples shown until this point the server responds to an event sent by the client. But for some applications, the server needs to be the originator of a message. This can be useful to send notifications to clients of events that originated in the server, for example in a background thread. The `socketio.send()` and `socketio.emit()` methods can be used to broadcast to all connected clients:

```
def some_function():
    socketio.emit('some event', {'data': 42})
```

Note that `socketio.send()` and `socketio.emit()` are not the same functions as the context-aware `send()` and `emit()`. Also note that in the above usage there is no client context, so `broadcast=True` is assumed and does not need to be specified.

CHAPTER 7

Rooms

For many applications it is necessary to group users into subsets that can be addressed together. The best example is a chat application with multiple rooms, where users receive messages from the room or rooms they are in, but not from other rooms where other users are. Flask-SocketIO supports this concept of rooms through the `join_room()` and `leave_room()` functions:

```
from flask_socketio import join_room, leave_room

@socketio.on('join')
def on_join(data):
    username = data['username']
    room = data['room']
    join_room(room)
    send(username + ' has entered the room.', room=room)

@socketio.on('leave')
def on_leave(data):
    username = data['username']
    room = data['room']
    leave_room(room)
    send(username + ' has left the room.', room=room)
```

The `send()` and `emit()` functions accept an optional `room` argument that cause the message to be sent to all the clients that are in the given room.

All clients are assigned a room when they connect, named with the session ID of the connection, which can be obtained from `request.sid`. A given client can join any rooms, which can be given any names. When a client disconnects it is removed from all the rooms it was in. The context-free `socketio.send()` and `socketio.emit()` functions also accept a `room` argument to broadcast to all clients in a room.

Since all clients are assigned a personal room, to address a message to a single client, the session ID of the client can be used as the room argument.

CHAPTER 8

Connection Events

Flask-SocketIO also dispatches connection and disconnection events. The following example shows how to register handlers for them:

```
@socketio.on('connect', namespace='/chat')
def test_connect():
    emit('my response', {'data': 'Connected'})

@socketio.on('disconnect', namespace='/chat')
def test_disconnect():
    print('Client disconnected')
```

The connection event handler can optionally return `False` to reject the connection. This is so that the client can be authenticated at this point.

Note that connection and disconnection events are sent individually on each namespace used.

CHAPTER 9

Error Handling

Flask-SocketIO can also deal with exceptions:

```
@socketio.on_error()        # Handles the default namespace
def error_handler(e):
    pass

@socketio.on_error('/chat') # handles the '/chat' namespace
def error_handler_chat(e):
    pass

@socketio.on_error_default  # handles all namespaces without an explicit error handler
def default_error_handler(e):
    pass
```

Error handler functions take the exception object as an argument.

The message and data arguments of the current request can also be inspected with the `request.event` variable, which is useful for error logging and debugging outside the event handler:

```
from flask import request

@socketio.on("my error event")
def on_my_event(data):
    raise RuntimeError()

@socketio.on_error_default
def default_error_handler(e):
    print(request.event["message"]) # "my error event"
    print(request.event["args"])    # (data,)
```

CHAPTER 10

Access to Flask's Context Globals

Handlers for SocketIO events are different than handlers for routes and that introduces a lot of confusion around what can and cannot be done in a SocketIO handler. The main difference is that all the SocketIO events generated for a client occur in the context of a single long running request.

In spite of the differences, Flask-SocketIO attempts to make working with SocketIO event handlers easier by making the environment similar to that of a regular HTTP request. The following list describes what works and what doesn't:

- An application context is pushed before invoking an event handler making `current_app` and `g` available to the handler.
- A request context is also pushed before invoking a handler, also making `request` and `session` available. But note that WebSocket events do not have individual requests associated with them, so the request context that started the connection is pushed for all the events that are dispatched during the life of the connection.
- The `request` context global is enhanced with a `sid` member that is set to a unique session ID for the connection. This value is used as an initial room where the client is added.
- The `request` context global is enhanced with `namespace` and `event` members that contain the currently handled namespace and event arguments. The `event` member is a dictionary with `message` and `args` keys.
- The `session` context global behaves in a different way than in regular requests. A copy of the user session at the time the SocketIO connection is established is made available to handlers invoked in the context of that connection. If a SocketIO handler modifies the session, the modified session will be preserved for future SocketIO handlers, but regular HTTP route handlers will not see these changes. Effectively, when a SocketIO handler modifies the session, a "fork" of the session is created exclusively for these handlers. The technical reason for this limitation is that to save the user session a cookie needs to be sent to the client, and that requires HTTP request and response, which do not exist in a SocketIO connection. When using server-side sessions such as those provided by the Flask-Session or Flask-KVSession extensions, changes made to the session in HTTP route handlers can be seen by SocketIO handlers, as long as the session is not modified in the SocketIO handlers.
- The `before_request` and `after_request` hooks are not invoked for SocketIO event handlers.
- SocketIO handlers can take custom decorators, but most Flask decorators will not be appropriate to use for a SocketIO handler, given that there is no concept of a `Response` object during a SocketIO connection.

CHAPTER 11

Authentication

A common need of applications is to validate the identify of their users. The traditional mechanisms based on web forms and HTTP requests cannot be used in a SocketIO connection, since there is no place to send HTTP requests and responses. If necessary, an application can implement a customized login form that sends credentials to the server as a SocketIO message when the submit button is pressed by the user.

However, in most cases it is more convenient to perform the traditional authentication process before the SocketIO connection is established. The user's identify can then be recorded in the user session or in a cookie, and later when the SocketIO connection is established that information will be accessible to SocketIO event handlers.

11.1 Using Flask-Login with Flask-SocketIO

Flask-SocketIO can access login information maintained by Flask-Login. After a regular Flask-Login authentication is performed and the `login_user()` function is called to record the user in the user session, any SocketIO connections will have access to the `current_user` context variable:

```
@socketio.on('connect')
def connect_handler():
    if current_user.is_authenticated:
        emit('my response',
             {'message': '{0} has joined'.format(current_user.name)},
             broadcast=True)
    else:
        return False  # not allowed here
```

Note that the `login_required` decorator cannot be used with SocketIO event handlers, but a custom decorator that disconnects non-authenticated users can be created as follows:

```
import functools
from flask import request
from flask_login import current_user
from flask_socketio import disconnect

def authenticated_only(f):
    @functools.wraps(f)
    def wrapped(*args, **kwargs):
        if not current_user.is_authenticated:
            disconnect()
        else:
            return f(*args, **kwargs)
    return wrapped
```

```
@socketio.on('my event')
@authenticated_only
def handle_my_custom_event(data):
    emit('my response', {'message': '{0} has joined'.format(current_user.name)},
         broadcast=True)
```

CHAPTER 12

Deployment

There are many options to deploy a Flask-SocketIO server, ranging from simple to the insanely complex. In this section, the most commonly used options are described.

12.1 Embedded Server

The simplest deployment strategy is to have eventlet or gevent installed, and start the web server by calling `socketio.run(app)` as shown in examples above. This will run the application on the eventlet or gevent web servers, whichever is installed.

Note that `socketio.run(app)` runs a production ready server when eventlet or gevent are installed. If neither of these are installed, then the application runs on Flask's development web server, which is not appropriate for production use.

12.2 Gunicorn Web Server

An alternative to `socketio.run(app)` is to use gunicorn as web server, using the eventlet or gevent workers. For this option, eventlet or gevent need to be installed, in addition to gunicorn. The command line that starts the eventlet server via gunicorn is:

```
gunicorn --worker-class eventlet -w 1 module:app
```

If you prefer to use gevent, the command to start the server is:

```
gunicorn -k gevent -w 1 module:app
```

When using gunicorn with the gevent worker and the WebSocket support provided by gevent-websocket, the command that starts the server must be changed to select a custom gevent web server that supports the WebSocket protocol. The modified command is:

```
gunicorn -k geventwebsocket.gunicorn.workers.GeventWebSocketWorker -w 1 module:app
```

In all these commands, `module` is the Python module or package that defines the application instance, and `app` is the application instance itself.

Gunicorn release 18.0 is the recommended release to use with Flask-SocketIO. The 19.x releases are known to have incompatibilities in certain deployment scenarios that include WebSocket.

Due to the limited load balancing algorithm used by gunicorn, it is not possible to use more than one worker process when using this web server. For that reason, all the examples above include the `-w 1` option.

12.3 uWSGI Web Server

At this time, uWSGI is not a good choice of web server for a SocketIO application due to the following limitations:

- The `'eventlet'` async mode cannot be used, as uWSGI currently does not support web servers based on eventlet.
- The `'gevent'` async mode is supported, but uWSGI is currently incompatible with the gevent-websocket package, so only the long-polling transport can be used.
- The native WebSocket support available from uWSGI is not based on eventlet or gevent, so it cannot be used at this time. If possible, a WebSocket transport based on the uWSGI WebSocket implementation will be made available in a future release.

12.4 Using nginx as a WebSocket Reverse Proxy

It is possible to use nginx as a front-end reverse proxy that passes requests to the application. However, only releases of nginx 1.4 and newer support proxying of the WebSocket protocol. Below is an example nginx configuration that proxies HTTP and WebSocket requests:

```
server {
    listen 80;
    server_name localhost;
    access_log /var/log/nginx/example.log;

    location / {
        proxy_pass http://127.0.0.1:5000;
        proxy_redirect off;

        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }

    location /socket.io {
        proxy_pass http://127.0.0.1:5000/socket.io;
        proxy_redirect off;
        proxy_buffering off;

        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "Upgrade";
    }
}
```

12.5 Using Multiple Workers

Flask-SocketIO supports multiple workers behind a load balancer starting with release 2.0. Deploying multiple workers gives applications that use Flask-SocketIO the ability to spread the client connections among multiple processes and hosts, and in this way scale to support very large numbers of concurrent clients.

There are two requirements to use multiple Flask-SocketIO workers:

- The load balancer must be configured to forward all HTTP requests from a given client always to the same worker. This is sometimes referenced as "sticky sessions". For nginx, use the `ip_hash` directive to achieve this. Gunicorn cannot be used with multiple workers because its load balancer algorithm does not support sticky sessions.
- Since each of the servers owns only a subset of the client connections, a message queue such as Redis or RabbitMQ is used by the servers to coordinate complex operations such as broadcasting and rooms.

When working with a message queue, there are additional dependencies that need to be installed:

- For Redis, the package `redis` must be installed (`pip install redis`).
- For RabbitMQ, the package `kombu` must be installed (`pip install kombu`).
- For other message queues supported by Kombu, see the Kombu documentation to find out what dependencies are needed.
- If eventlet or gevent are used, then monkey patching the Python standard library is normally required to force the message queue package to use coroutine friendly functions and classes.

To start multiple Flask-SocketIO servers, you must first ensure you have the message queue service running. To start a Socket.IO server and have it connect to the message queue, add the `message_queue` argument to the `SocketIO` constructor:

```
socketio = SocketIO(app, message_queue='redis://')
```

The value of the `message_queue` argument is the connection URL of the queue service that is used. For a redis queue running on the same host as the server, the `'redis://'` URL can be used. Likewise, for a default RabbitMQ queue the `'amqp://'` URL can be used. The Kombu package has a documentation section that describes the format of the URLs for all the supported queues.

12.6 Emitting from an External Process

For many types of applications, it is necessary to emit events from a process that is not the SocketIO server, for a example a Celery worker. If the SocketIO server or servers are configured to listen on a message queue as shown in the previous section, then any other process can create its own `SocketIO` instance and use it to emit events in the same way the server does.

For example, for an application that runs on an eventlet web server and uses a Redis message queue, the following Python script broadcasts an event to all clients:

```
socketio = SocketIO(message_queue='redis://')
socketio.emit('my event', {'data': 'foo'}, namespace='/test')
```

When using the `SocketIO` instance in this way, the Flask application instance is not passed to the constructor.

The `channel` argument to `SocketIO` can be used to select a specific channel of communication through the message queue. Using a custom channel name is necessary when there are multiple independent SocketIO services sharing the same queue.

Flask-SocketIO does not apply monkey patching when eventlet or gevent are used. But when working with a message queue, it is very likely that the Python package that talks to the message queue service will hang if the Python standard library is not monkey patched.

It is important to note that an external process that wants to connect to a SocketIO server does not need to use eventlet or gevent like the main server. Having a server use a coroutine framework, while an external process does not is not a problem. For example, Celery workers do not need to be configured to use eventlet or gevent just because the main

server does. But if your external process does use a coroutine framework for whatever reason, then monkey patching is likely required, so that the messaque queue accesses coroutine friendly functions and classes.

CHAPTER 13

API Reference

class flask_socketio.**SocketIO**(*app=None*, ***kwargs*)

Create a Flask-SocketIO server.

Parameters

- **app** – The flask application instance. If the application instance isn't known at the time this class is instantiated, then call `socketio.init_app(app)` once the application instance is available.
- **message_queue** – A connection URL for a message queue service the server can use for multi-process communication. A message queue is not required when using a single server process.
- **channel** – The channel name, when using a message queue. If a channel isn't specified, a default channel will be used. If multiple clusters of SocketIO processes need to use the same message queue without interfering with each other, then each cluster should use a different channel.
- **resource** – The SocketIO resource name. Defaults to `'socket.io'`. Leave this as is unless you know what you are doing.
- **kwargs** – Socket.IO and Engine.IO server options.

The Socket.IO server options are detailed below:

Parameters

- **client_manager** – The client manager instance that will manage the client list. When this is omitted, the client list is stored in an in-memory structure, so the use of multiple connected servers is not possible. In most cases, this argument does not need to be set explicitly.
- **logger** – To enable logging set to `True` or pass a logger object to use. To disable logging set to `False`.
- **binary** – `True` to support binary payloads, `False` to treat all payloads as text. On Python 2, if this is set to `True`, `unicode` values are treated as text, and `str` and `bytes` values are treated as binary. This option has no effect on Python 3, where text and binary payloads are always automatically discovered.
- **json** – An alternative json module to use for encoding and decoding packets. Custom json modules must have `dumps` and `loads` functions that are compatible with the standard library versions.

The Engine.IO server configuration supports the following settings:

Parameters

- **async_mode** – The library used for asynchronous operations. Valid options are "threading", "eventlet" and "gevent". If this argument is not given, "eventlet" is tried first, then "gevent", and finally "threading". The websocket transport is not supported in "threading" mode.
- **ping_timeout** – The time in seconds that the client waits for the server to respond before disconnecting.
- **ping_interval** – The interval in seconds at which the client pings the server.
- **max_http_buffer_size** – The maximum size of a message when using the polling transport.
- **allow_upgrades** – Whether to allow transport upgrades or not.
- **http_compression** – Whether to compress packages when using the polling transport.
- **compression_threshold** – Only compress messages when their byte size is greater than this value.
- **cookie** – Name of the HTTP cookie that contains the client session id. If set to `None`, a cookie is not sent to the client.
- **cors_allowed_origins** – List of origins that are allowed to connect to this server. All origins are allowed by default.
- **cors_credentials** – Whether credentials (cookies, authentication) are allowed in requests to this server.
- **engineio_logger** – To enable Engine.IO logging set to `True` or pass a logger object to use. To disable logging set to `False`.

on(*message*, *namespace=None*)

Decorator to register a SocketIO event handler.

This decorator must be applied to SocketIO event handlers. Example:

```
@socketio.on('my event', namespace='/chat')
def handle_my_custom_event(json):
    print('received json: ' + str(json))
```

Parameters

- **message** – The name of the event. This is normally a user defined string, but a few event names are already defined. Use `'message'` to define a handler that takes a string payload, `'json'` to define a handler that takes a JSON blob payload, `'connect'` or `'disconnect'` to create handlers for connection and disconnection events.
- **namespace** – The namespace on which the handler is to be registered. Defaults to the global namespace.

on_error(*namespace=None*)

Decorator to define a custom error handler for SocketIO events.

This decorator can be applied to a function that acts as an error handler for a namespace. This handler will be invoked when a SocketIO event handler raises an exception. The handler function must accept one argument, which is the exception raised. Example:

```
@socketio.on_error(namespace='/chat')
def chat_error_handler(e):
    print('An error has occurred: ' + str(e))
```

Parameters **namespace** – The namespace for which to register the error handler. Defaults to the global namespace.

on_error_default(*exception_handler*)

Decorator to define a default error handler for SocketIO events.

This decorator can be applied to a function that acts as a default error handler for any namespaces that do not have a specific handler. Example:

```
@socketio.on_error_default
def error_handler(e):
    print('An error has occurred: ' + str(e))
```

emit(*event*, **args*, ***kwargs*)

Emit a server generated SocketIO event.

This function emits a SocketIO event to one or more connected clients. A JSON blob can be attached to the event as payload. This function can be used outside of a SocketIO event context, so it is appropriate to use when the server is the originator of an event, outside of any client context, such as in a regular HTTP request handler or a background task. Example:

```
@app.route('/ping')
def ping():
    socketio.emit('ping event', {'data': 42}, namespace='/chat')
```

Parameters

- **event** – The name of the user event to emit.
- **args** – A dictionary with the JSON data to send as payload.
- **namespace** – The namespace under which the message is to be sent. Defaults to the global namespace.
- **room** – Send the message to all the users in the given room. If this parameter is not included, the event is sent to all connected users.
- **include_self** – `True` to include the sender when broadcasting or addressing a room, or `False` to send to everyone but the sender.
- **callback** – If given, this function will be called to acknowledge that the client has received the message. The arguments that will be passed to the function are those provided by the client. Callback functions can only be used when addressing an individual client.

send(*data*, *json=False*, *namespace=None*, *room=None*, *callback=None*, *include_self=True*)

Send a server-generated SocketIO message.

This function sends a simple SocketIO message to one or more connected clients. The message can be a string or a JSON blob. This is a simpler version of `emit()`, which should be preferred. This function can be used outside of a SocketIO event context, so it is appropriate to use when the server is the originator of an event.

Parameters

- **message** – The message to send, either a string or a JSON blob.
- **json** – `True` if `message` is a JSON blob, `False` otherwise.

- **namespace** – The namespace under which the message is to be sent. Defaults to the global namespace.
- **room** – Send the message only to the users in the given room. If this parameter is not included, the message is sent to all connected users.
- **include_self** – `True` to include the sender when broadcasting or addressing a room, or `False` to send to everyone but the sender.
- **callback** – If given, this function will be called to acknowledge that the client has received the message. The arguments that will be passed to the function are those provided by the client. Callback functions can only be used when addressing an individual client.

close_room(*room*, *namespace=None*)
Close a room.

This function removes any users that are in the given room and then deletes the room from the server. This function can be used outside of a SocketIO event context.

Parameters

- **room** – The name of the room to close.
- **namespace** – The namespace under which the room exists. Defaults to the global namespace.

run(*app*, *host=None*, *port=None*, ***kwargs*)
Run the SocketIO web server.

Parameters

- **app** – The Flask application instance.
- **host** – The hostname or IP address for the server to listen on. Defaults to 127.0.0.1.
- **port** – The port number for the server to listen on. Defaults to 5000.
- **debug** – `True` to start the server in debug mode, `False` to start in normal mode.
- **use_reloader** – `True` to enable the Flask reloader, `False` to disable it.
- **extra_files** – A list of additional files that the Flask reloader should watch. Defaults to `None`
- **log_output** – If `True`, the server logs all incomming connections. If `False` logging is disabled. Defaults to `True` in debug mode, `False` in normal mode. Unused when the threading async mode is used.
- **kwargs** – Additional web server options. The web server options are specific to the server used in each of the supported async modes. Note that options provided here will not be seen when using an external web server such as gunicorn, since this method is not called in that case.

stop()
Stop a running SocketIO web server.

This method must be called from a HTTP or SocketIO handler function.

test_client(*app*, *namespace=None*)
Return a simple SocketIO client that can be used for unit tests.

flask_socketio.**emit**(*event*, **args*, ***kwargs*)
Emit a SocketIO event.

This function emits a SocketIO event to one or more connected clients. A JSON blob can be attached to the event as payload. This is a function that can only be called from a SocketIO event handler, as in obtains some information from the current client context. Example:

```
@socketio.on('my event')
def handle_my_custom_event(json):
    emit('my response', {'data': 42})
```

Parameters

- **event** – The name of the user event to emit.
- **args** – A dictionary with the JSON data to send as payload.
- **namespace** – The namespace under which the message is to be sent. Defaults to the namespace used by the originating event. A `'/'` can be used to explicitly specify the global namespace.
- **callback** – Callback function to invoke with the client's acknowledgement.
- **broadcast** – `True` to send the message to all clients, or `False` to only reply to the sender of the originating event.
- **room** – Send the message to all the users in the given room. If this argument is set, then broadcast is implied to be `True`.
- **include_self** – `True` to include the sender when broadcasting or addressing a room, or `False` to send to everyone but the sender.

flask_socketio.**send**(*message*, ***kwargs*)

Send a SocketIO message.

This function sends a simple SocketIO message to one or more connected clients. The message can be a string or a JSON blob. This is a simpler version of `emit()`, which should be preferred. This is a function that can only be called from a SocketIO event handler.

Parameters

- **message** – The message to send, either a string or a JSON blob.
- **namespace** – The namespace under which the message is to be sent. Defaults to the namespace used by the originating event. An empty string can be used to use the global namespace.
- **callback** – Callback function to invoke with the client's acknowledgement.
- **broadcast** – `True` to send the message to all connected clients, or `False` to only reply to the sender of the originating event.
- **room** – Send the message to all the users in the given room.
- **include_self** – `True` to include the sender when broadcasting or addressing a room, or `False` to send to everyone but the sender.

flask_socketio.**join_room**(*room*)

Join a room.

This function puts the user in a room, under the current namespace. The user and the namespace are obtained from the event context. This is a function that can only be called from a SocketIO event handler. Example:

```
@socketio.on('join')
def on_join(data):
    username = session['username']
```

```
        room = data['room']
        join_room(room)
        send(username + ' has entered the room.', room=room)
```

Parameters **room** – The name of the room to join.

flask_socketio.**leave_room**(*room*)
Leave a room.

This function removes the user from a room, under the current namespace. The user and the namespace are obtained from the event context. This is a function that can only be called from a SocketIO event handler. Example:

```
    @socketio.on('leave')
    def on_leave(data):
        username = session['username']
        room = data['room']
        leave_room(room)
        send(username + ' has left the room.', room=room)
```

Parameters **room** – The name of the room to leave.

flask_socketio.**close_room**(*room*)
Close a room.

This function removes any users that are in the given room and then deletes the room from the server. This is a function that can only be called from a SocketIO event handler.

Parameters **room** – The name of the room to close.

flask_socketio.**rooms**()
Return a list of the rooms the client is in.

This function returns all the rooms the client has entered, including its own room, assigned by the Socket.IO server. This is a function that can only be called from a SocketIO event handler.

flask_socketio.**disconnect**(*silent=False*)
Disconnect the client.

This function terminates the connection with the client. As a result of this call the client will receive a disconnect event. Example:

```
    @socketio.on('message')
    def receive_message(msg):
        if is_banned(session['username']):
            disconnect()
        # ...
```

Parameters **silent** – this option is deprecated.

f

C

D

E

F

J

L

O

R

S

T